Lessons Learned from AlphaGo

Steffen Hölldobler[1,2]
sh@iccl.tu-dresden.de

Sibylle Möhle[1]
sibylle.moehle@tu-dresden.de

Anna Tigunova[1*]
belpku001@gmail.com

[1]International Center for Computational Logic, TU Dresden, 01062 Dresden, Germany
[2]North-Caucasus Federal University, Stavropol, Russian Federation

Abstract

The game of Go is known to be one of the most complicated board games. Competing in Go against a professional human player has been a long-standing challenge for AI. In this paper we shed light on the AlphaGo program that could beat a Go world champion, which was previously considered non-achievable for the state of the art AI.

1 Introduction

The game of Go has been around for centuries and it is indeed a very popular brainteaser nowadays. Along with Chess and Checkers, Go is a game of perfect information. That is, the outcome of the game solely depends on the strategy of both players. This makes it attractive to solve Go computationally because we can rely on a machine to find the optimal sequence of moves. However, this task is extremely difficult due to the huge search space of possible moves. Therefore, Go has been considered a desired frontier for AI, which was predicted to be not achievable in the next decade [BC01].

Until recent time, many computer Go bots appeared, still they barely achieved the level of a master player, let alone play on a par with the professionals [RTL+10]. However, in the beginning of 2016, Google DeepMind published an article, where they stated that their program, AlphaGo, was able to win over a professional player [SHM+16]. Several months after that AlphaGo defeated the world Go champion in an official match, an event of a great importance, because now the "grand challenge" [Mec98, CI07] was mastered.

The problem with Go is the size of the board, which yields a 10^{170} positions state space [MHSS14, VDHUVR02]. In comparison, Chess' state space is about 10^{43} [Chi96]. Such games are known to have a high *branching factor* - the number of available moves from the current position. The number of possible game scenarios in Go is greater than the number of atoms in the universe [TF06].

The authors of AlphaGo managed to solve this problem. The system they designed is based on tree search, boosted by neural networks predicting the moves. However, all these techniques are not novel in computer Go and have been utilized by other authors, too.

So what makes AlphaGo so special? In our paper we address this question. Here we discuss how AlphaGo is designed in the context of the history of computer Go. By unfolding the architecture of AlphaGo we show that every single detail of its implementation is a result of many years' research, but their ensemble is the key to AlphaGo's success.

The rest of the paper is structured as follows. First, we provide an overview of Go. After that we show how the state of the art research tried to solve Go, followed by the description of AlphaGo's approach.

*The authors are mentioned in this paper is the alphabetical order.

In: S. Hölldobler, A. Malikov, C. Wernhard (eds.): *YSIP2 – Proceedings of the Second Young Scientist's International Workshop on Trends in Information Processing, Dombai, Russian Federation, May 16–20, 2017*, published at http://ceur-ws.org.

2 The Game of Go

2.1 The Rules

Go originated in China about 2-5 thousand of years ago and was later spread to Japan and Korea. The first sources in Europe mentioning Go date back to the 16th century. Now Go is played all over the world, it has one of the biggest number of players.

Go is played on a 19×19 board which is essentially a grid. There are two players in the game, one plays black and the other plays white stones. At the beginning, the board is empty and the black player starts by placing his stone in one of the grid intersections. The white player places his stone in an unoccupied grid point. Players can not move the stones, only put them on the board, taking turns. However, a player can skip his move, allowing the other one to play twice. The aim of each player is to occupy the most of the territory with his stones.

Each stone has *liberties* - free adjacent cells. The adjacent stones of the same color form a *group*. A player can capture a group of the opponent's stones by surrounding it with his own stones. Then that whole group is removed from the board.

The game ends after both players passed consecutively, and then the scores are counted. The system of counting points can differ. Generally, the points are awarded to the player proportionally to the number of grid points occupied by this player and the empty points surrounded with player's stones. The basic aim of the player is to place as many stones as possible, capturing the opponent's groups and protecting his own ones. We refer to [Mül02] for more details on Go rules.

2.2 The Challenge of Go

Go is a difficult game due to the following two reasons. First, Go has a large number of moves, which yields exponentially many ways the game may unfold. Secondly, it is very difficult to understand precisely who is winning in the given position. It limits the ability of using Machine Learning approaches. The attempts to build a computer Go player have been exercised since the 1970s [BC01].

3 Playing Go with Monte Carlo Tree Search

In this section we describe techniques that could be applied by computers to play Go. We start by giving details on how generally computers can play board games.

3.1 Game Tree

Let us consider finite two-player games (such as Go, Chess, etc.). These games are semantically represented by a *game tree*. The game tree is a directed acyclic graph. Each node in the graph represents a state in the game, each edge is a move of the player. The root of the tree is the initial state of the board. By descending the game tree starting from the root, one can represent the game, move by move. The leaf nodes denote the terminal states in the game (win/lose/draw).

The complete game tree contains all possible moves from each state. In the complete game tree the number of leaves is the number of all existing game scenarios. The *branching factor* of the game is the number of states at the horizontal level. Each level is a *ply* - a single move by one of the players from the current position. The depth of the tree reflects how many moves should be made before the end of the game is reached.

The game strategy is determined by the search of the game tree. The moves that ensure the victory at the end of the game should be picked. The search is generally done by the *minimax* algorithm, which tries to minimize the possible loss under the perfect play of the opponent [KM75]. By these means the player maximizes his reward while minimizing the reward of the opponent.

When the size of the game is relatively small (like for Tic-Tac-Toe), it is feasible to search the complete game tree using minimax. However, when the game grows in size, the complete tree becomes infeasible [BC01]. To tackle this, several search strategies exist that can be divided into two categories: selective and non-selective search algorithms [Upt98].

- **Non-selective** search considers all moves and searches through them equally. The major disadvantage is that the algorithms consider a lot of useless moves, which could have been thrown out of consideration from scratch. The examples of such techniques are *alpha-beta pruning* [KM75] and *iterative deepening* [Kor85]. Alpha-beta pruning led to the considerable advances in search heuristics development [Upt98], it enabled computers to solve a larger number of board-games; however, it is still too weak for the size of the Go game tree. Another problem with *pruning* is that in general it requires domain-specific knowledge. Iterative

deepening also suffers from the fact that too many nodes have to be visited, basically all on each level of the tree.

- **Selective** search techniques use prior estimations to choose the most promising move to explore. They model the way humans make moves in the games – using game domain knowledge and pattern matching [CS15], they only look into the most profitable moves. Therefore, the effort in the selective approach is concentrated on picking the right function that estimates the worthiness of the moves. One of these search techniques is the *Monte Carlo method*.

3.2 The Monte Carlo Method

One of the ways to efficiently tackle the game search tree is to apply the *Monte Carlo method* [Fis96]. This method approximates a function with the use of random sampling. The mean of the samples converges to the real value of a function. Monte Carlo is commonly used when the problem is too large to be solved analytically. It has been widely applied in mathematics, statistics and further disciplines.

The first Go program using the Monte Carlo method was *Gobble* [Brü93], which was back in 1993. Gobble used the average of many simulations to assign the approximate value to possible moves and then considered only the moves with the highest values. However, the rank of Gobble was 25 *kyu*[1] - a very weak beginner.

3.3 The Monte Carlo Tree Search

The idea to incorporate Monte Carlo simulations into the tree growing process introduced in [Jui99] was first adopted for Go by [Cou06] in the program *Crazy Stone*. This program has managed to outperform the conventional techniques and since then the Monte Carlo Tree Search (MCTS) has enjoyed great popularity in the computer Go community.

Briefly, in MCTS during the descent each move is chosen according to its value, which is accumulated by making random simulations, each one representing a complete game. The value for the move reflects the information of the number and the outcome of the simulations that have ran through it.

This approximation is justified by a central limit theorem, which says that the Monte Carlo values (mean of all outcomes) converge to the normal distribution. If the tree is explored to a fair extent, the strategy using MCTS converges to the optimal strategy [KS06a].

MCTS has an important advantage over pruning [Bou06]. The values of the moves may change as the tree grows and initially inferior moves may eventually gain importance. In pruning, the branches are completely discarded, in contrast to MCTS, where they still have non-zero probability and a chance to come into play later.

Let us now look into MCTS in greater detail, as it is the main building block of AlphaGo's algorithms.

3.3.1 Stages of Monte Carlo Tree Search

The state-of-the-art form of MCTS consists of four stages that are applied sequentially: selection, expansion, simulation and backpropagation [CBSS08]. These stages are reflected in Figure 1.

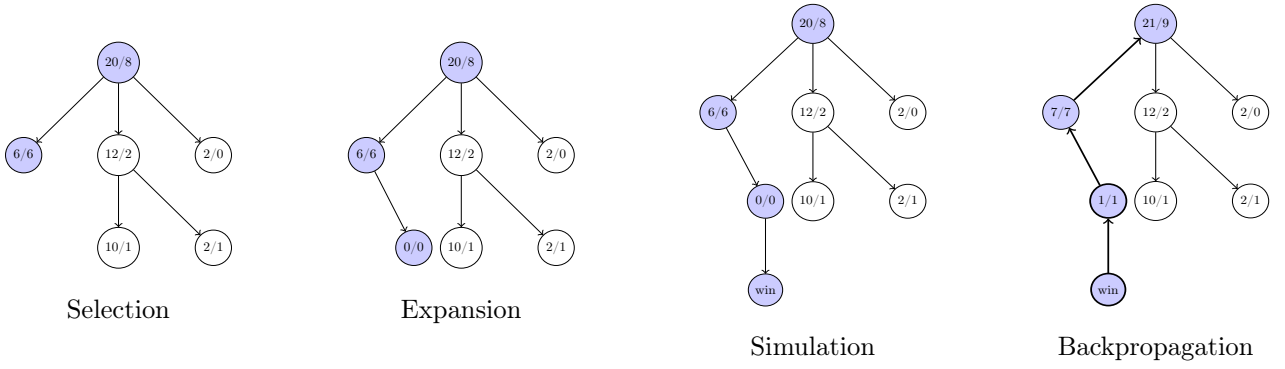

Figure 1: Monte Carlo Tree Search. In the selection phase the scores for all child nodes are known and the greatest one is chosen. In the expansion phase a new node is added to the tree. Random simulations from the new node are run in the simulation phase. The result of the simulation is returned to all the nodes in the path during backpropogation. The numbers in the nodes denote all_visits/winning_visits count.

[1]Kyu is a student rank in the Go player ranking system, with the lowest kyu being 30.

Selection In the *selection* stage the current node is within the explored part of the search tree and all values for the following actions are known. In this stage the algorithm selects the action according to these values keeping a balance between exploitation (selecting the actions with highest win rate) and exploration (trying out less explored moves).
Expansion Once we get to a node which has not been encountered so far, this node is added to the tree with a randomly initialized value.
Simulation After adding a new node, we need to refine the random value that has been assigned to it. This is done by randomly simulating the rest of the game to the terminal point. This is also called a *rollout phase*.
Backpropagation When the rollout reaches the end of the game, we get the final reward. Then we go up the tree and update the values in each parent node. First, we increment the visit count and add the reward.

3.3.2 Advantages of MCTS

The main advantage of MCTS is that it dramatically reduces the number of explored nodes in the tree. The search tree grows asymmetrically with a low branching factor allowing only for the promising branches [CM07].

Secondly, MCTS requires no domain knowledge. Thus, MCTS is very popular for general game playing - when the rules are not known. But then, MCTS can be tailored to a certain application by inserting some domain-specific heuristics [DU07].

Moreover, there is no need to have a fixed number of simulations, in contrast to *iterative deepening* [Bou04]. The search can be stopped after arbitrarily many time steps, and the best result achieved so far is returned. This is profitable when the time for the move in the game is limited and it is necessary to return some approximation for the optimal move.

In addition to that, Monte Carlo tree search is easy to parallelize because the simulations are independent. This advantage was utilized in the distributed version of AlphaGo [SHM+16].

3.4 Enhancements

Although MCTS is efficient in terms of reducing the search in the tree and has a profound theoretical background, it is still too weak to efficiently tackle the Go domain [KSW06]. Here a potential problem arises: the thinking time for the program is limited and it simply can not gather enough statistics to precisely approximate the optimal values for the nodes.

Therefore, MCTS has to be aided with some additional refinements. A good survey of different MCTS techniques is provided in [BPW+12]. Basically, the refinements can be added to several steps of MCTS.
Selection A good rule for choosing the actions in the selection stage needs to mix exploration and exploitation. One of the most often used rules is UCB [KS06a] (Upper Confidence Bound). The original variant of it is represented with a formula:

$$\pm(x_i + \sqrt{\frac{2 \ln n}{n_i}}) \tag{1}$$

Here, x_i is an average reward from the ith choice, n_i is the number of times this choice was taken and n is the total number of tries. The strategy is to pick the move with the greatest upper bound of the value estimated in (1). The first term in Equation (1) is responsible for exploitation, the second for exploration.

Formula 1 is efficiently used in the UCT algorithm (UCB applied to trees) introduced in [KS06a]. Also, in the aforementioned paper MCTS with UCT was proven to converge to minimax given enough simulations.
Expansion In the expansion step the value for the newly added node can be initialized not randomly but according to the *value function* [GS07]. The value function $v : S \to \{\pm 1\}$ determines the outcome of the game starting from the current state (see Section 4.7).
Simulation Random rollouts are very unrealistic and provide a poor evaluation of the target function. The idea is to make them more sensible, either by using some domain knowledge or by learning tactics of the game. One example of the former is a *rule-based* approach [ST09]. Some examples of learning are *temporal difference learning* [FB10] and *pattern matching* [WG07, DU07, Cou07], the last being arguably the most popular approach.

There are still more enhancements to MCTS in the state-of-the-art research [BPW+12, Bai10]. Now let us take a look at the similar enhancements that were implemented within AlphaGo and which arguably enabled it to achieve such amazing performance.

4 Deep Reinforcement Learning to Support MCTS

To enhance the performance of MCTS AlphaGo utilizes Deep Convolutional Neural Networks and Reinforcement Learning. In this section we first provide the background of these approaches and later discuss how they are applied in the MCTS settings.

4.1 Motivation

The Monte Carlo method has the nice property that the mean value of simulations converges to the true value; however to guarantee this, it should be provided with an infinite number of simulations. Of course, in no domain is this possible, let alone board games, where the thinking time is strictly limited [KS06b].

Therefore, if the quantity of the simulations is limited, one has to increase the quality of every single simulation. The more realistic the decisions are, the faster convergence. The side-effect of the reduced number of simulations can be poorer *selection* which will be prone to overlooking the good moves.

For tackling this problem one may refer to the way humans play Go. Clearly, they also pick a move and try to predict the outcome after playing it. But contrarily to MCTS, people do this not absolutely randomly. The experienced players use *intuition* to preselect an action [CS15]. Of course, if MCTS was guided by this intuition, it would converge faster. Therefore, many authors [Dah99, CS15, SN08] try to imitate the way people think. One of the possible means is an artificial neural network.

4.2 Artificial Neural Networks

The simplest version of the neuron in the human brain is the *perceptron model* [Fra57], depicted in Figure (2). As an input it gets environmental information and computes a weighted sum of these inputs, which is called a *potential* of the perceptron. After that, the *activation function* is applied to the potential which is a simple threshold in the case of the perceptron: $sign(\sum w_i x_i + b)$. The result of the output function basically shows if the perceptron reacted to the external impulse. The weights w_i and the bias b are the trainable parameters of this linear model.

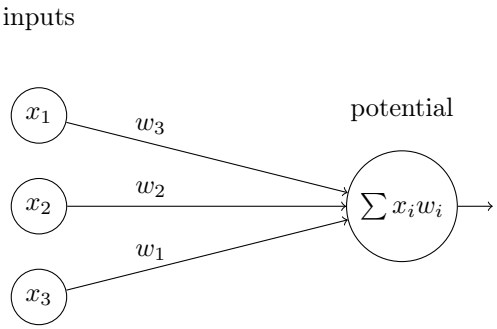

Figure 2: The perceptron

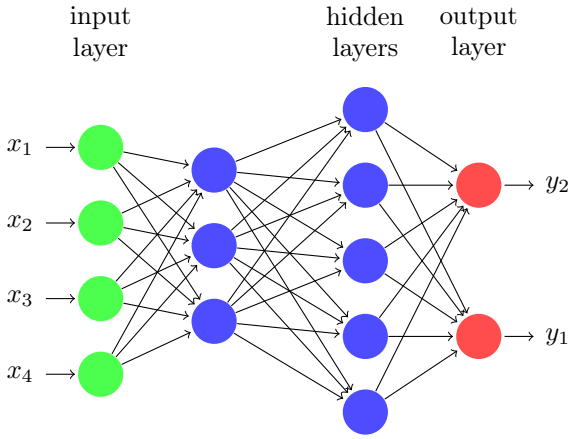

Figure 3: The multilayer feed-forward neural network

The output function can be replaced by a differentiable one. The weights of the neuron with this function can be trained by the gradient method (minimizing the loss by making the derivative 0). The most popular output functions are *sigmoid* and *tanh*, their advantage is the ability to react to the slight changes in the input [HM95].

The neuron is the principal unit in the *Artificial Neural Network* (ANN) [MP43], which is a quite crude approximate model of the human brain. The ANN is a bunch of neurons computing a mapping $f : \mathbb{R}^m \to \mathbb{R}^n$. There exists a wealth of different kinds of ANNs, but in our paper we would like to discuss only *feed-forward* NNs (3), as they are employed in AlphaGo.

In the feed-forward ANNs the neurons are arranged in layers, the connections exist only between consequent layers and there are no loops in the network. It consists of a single input and a single output layer and several hidden layers. The weights of the network are trained by *backpropagation* [WH86].

Feed-forward neural networks were applied to Go [Dah99]. However, due to the huge size of the Go board, it is highly difficult to incorporate it to the feed-forward neural network. For instance, if every intersection of

the board was fed into a separate input neuron, it would yield about 100k connections to the first hidden layer. Moreover, in a feed-forward neural network each neuron on the current level is connected with each neuron in the previous level. Every connection has its weight, which means that a lot of parameters have to be estimated.

To tackle this, one can assume that the most informative part of the board is within a limited space around the previous move - a current local "fight". Then it makes no sense to consider the whole board. This assumption was taken over in [Dah99].

Still, one can go further and introduce a whole new architecture of the neural network, that will take this idea into account. Here we are talking about *convolutional neural networks* that were implemented in AlphaGo.

4.3 Convolutional Neural Networks

In recent years the convolutional neural networks (CNNs) have been successfully applied in the domains of face and character recognition, image classification and other visual tasks [MHSS14]. Their operation is based on extracting local features from the input.

A convolutional neural network is based on two concepts: *local receptive fields* and *weight sharing*. The former concept means that each neuron is connected only to a small region of a fixed size in the previous layer. Every neuron is responsible to its own subarea in the previous layer, but the set of weights of connections, known as *filter*, is the same for all neurons (*weight sharing* assumption).

Each filter can be considered as a single feature extractor. The more different filters are applied, the more information we get from the input (256 filters are used in AlphaGo). The learning task for CNN is to find the weights within each filter.

One can observe that due to the fact that the filter is the same for all neurons in one layer and the size of the filter is restricted, the number of free parameters in the network is dramatically reduced [SN08].

CNNs have been very successfully applied in Go [MHSS14, SN08, CS15]. Due to the *translational invariance* [CS15] (the assessment of a position is not affected by shifting the board) and the local character of features in Go, CNNs come as the best choice.

4.4 Neural Networks in AlphaGo

Recall that the original aim of using neural networks was to make a resemblance of how humans think while playing Go. AlphaGo uses neural networks to *predict human moves*. For this reason the input to the AlphaGo's CNN is the current board setting and the output is the prediction of a move a person would make.

Let us describe it more precisely. To train the CNN, the authors took 30.000 games of Go professionals that were recorded on the Go server KGS [Sch10]. From each game random positions were selected together with the consequent actions of the player. This action was the objective of the network's prediction.

The input position was translated into 48 features, which indicate the color of the stone at each intersection, the number of free adjacent cells and some other information. These features have been selected according to the results of the previous research [CS15].

The input layer was, therefore, a $19\times19\times48$ stack, carrying the value of every feature for each intersection. The CNN had 13 hidden layers with 256 filters on each layer. The output layer was of the size 19×19 and each cell in the output contained the probability that a person will put a stone in the corresponding intersection.

The neural network was trained by a standard backpropagation [WH86]. The scheme above represents a *supervised learning* approach, therefore, we refer to the resulting network as the SL network. However, in AlphaGo also *reinforcement learning* was used.

4.5 Reinforcement Learning of the Neural Networks

Even if the quality of the move prediction was 100% (and the accuracy of AlphaGo's CNN prediction was reported to be 57% in [SHM$^+$16]), it will still be a score in the task of modeling peoples' moves. But the ultimate task is to win over human, therefore, some approach should be utilized that would enable AlphaGo to surpass the human abilities. To achieve this aim the *Reinforcement Learning* [Wil92] was applied in AlphaGo.

Conceptually, reinforcement learning is a method of training an AI agent by not giving him explicitly the correct answer. Instead, the aim of the agent is to maximize the *reward*, which is a symbolical environment feedback to his actions.

Getting back to the setting of Go, the reward is expressed as ±1 for win/lose in the end of the game and 0 for all other steps. Now we iteratively increase the power of the SL network by letting it play against the other versions of itself and consequently learn from its own games.

At one iteration step the current version (in terms of current parameters) is opposed by some random instance of the SL network from the previous iterations. They play a match of Go and get the final reward. This reward

tells the current SL network if its actions were correct and how the parameters should be accordingly adjusted. Every 500 iterations the version with the current parameters is added to the pool of previous versions. The reinforcement learning technique provided a significant improvement to the playing strength of the SL network.

4.6 Integrating Neural Networks with MCTS

So what are the neural networks used in AlphaGo for?

The SL network is used at the *selection* stage of MCTS to encourage exploration. Recall, that a good selection rule keeps a balance between selecting optimal known moves and investigating the new ones. AlphaGo uses a variant of the UCT rule to select the action a, maximizing the formula $x(a) + u(a)$, where $x(a)$ is the value of the move (which can be found with Monte Carlo rollouts) and $u(a)$ is proportional to $P(a)$ - the probability predicted by the SL neural network. In a sense, CNN biases MCTS to try out the moves which have been scarcely explored but which seem optimal to the CNN.

Although the reinforcement learning networks proved to be stronger than the SL networks, the overall performance of AlphaGo was better when the move selection was enhanced with the SL network predictions. It can be explained by the fact that the SL-network is more human-like, as it was trained on the real people's games [SHM+16]. People tend to explore more, either because of the mistakes during the game or out of ardor.

Nevertheless, the reinforcement learning networks (RL networks) found their application in the other component of AlphaGo, the *value network*, which is used to approximate the *value function*.

4.7 Value Networks

Let s denote a state in the game, then $v(s)$ is a *value function* ($v : S \to \pm 1$, where S is a state space). The value function predicts the game result (win/lose) for the current state s under the perfect play of both opponents.

Theoretically, value function can be found by the full traversal of the game tree, but as it was mentioned before, for Go it is not feasible. On the one hand, it is known that the mean value of the Monte Carlo rollouts converges to the value function [KS06b]. However, when too few simulations are made (due to the limited time for a move), the rollout results tend to be inaccurate [HM13].

To aid them, AlphaGo learns to approximate the value function by a powerful *value network*. This network has absolutely the same architecture as the SL network described above, however, given a position in the game as input, it outputs a single value, denoting win or loss. Another difference is that the value network is trained not on the human games, but on the reinforcement learning network's games. The idea to approximate the value function utilizing reinforcement learning was also presented in [Lit94].

Therefore, the value, determining selection in the first stage of MCTS in AlphaGo, is a mix of both value network estimations and the Monte Carlo rollouts' results. Moreover, the rollouts were also improved in AlphaGo.

4.8 Guided Rollouts

So far we have not discussed, how AlphaGo aids the simuation stage of MCTS. The quality of rollouts can dramatically strengthen MCTS [SHM+16].

In Section 3.4 we have noted a few possible approaches, such as *TD learning* and *pattern matching*. AlphaGo uses a relatively simple heuristic to guide rollouts: a linear classifier.

It works as follows: a small area around each legal move is taken and compared to one of the precomputed patterns. The input to the classifier is the array of indicators, saying that the area matches a particular pattern. The output of the classifier is the probability of the move being played. The classifier is again trained on the positions from KGS matches.

The advantage of the classifier described above is that it requires about 1000 times less computations than the neural network. It is very efficient when one needs to execute a great number of simulations in the limited time period.

4.9 Learning Pipeline

To put it all together, the architecture of AlphaGo consists of two distinct processes: *learning* and *playing*. The former is an offline stage, during which the training of the neutral networks is performed. The output of this stage is the SL, RL and value network and rollout policy as well. The overview is in Figure 5.

The second online stage is actually playing Go by traversing the game tree using MCTS. The search is enhanced by the predictors described above. The distinct stages of MCTS in AlphaGo are depicted in Figure 4.

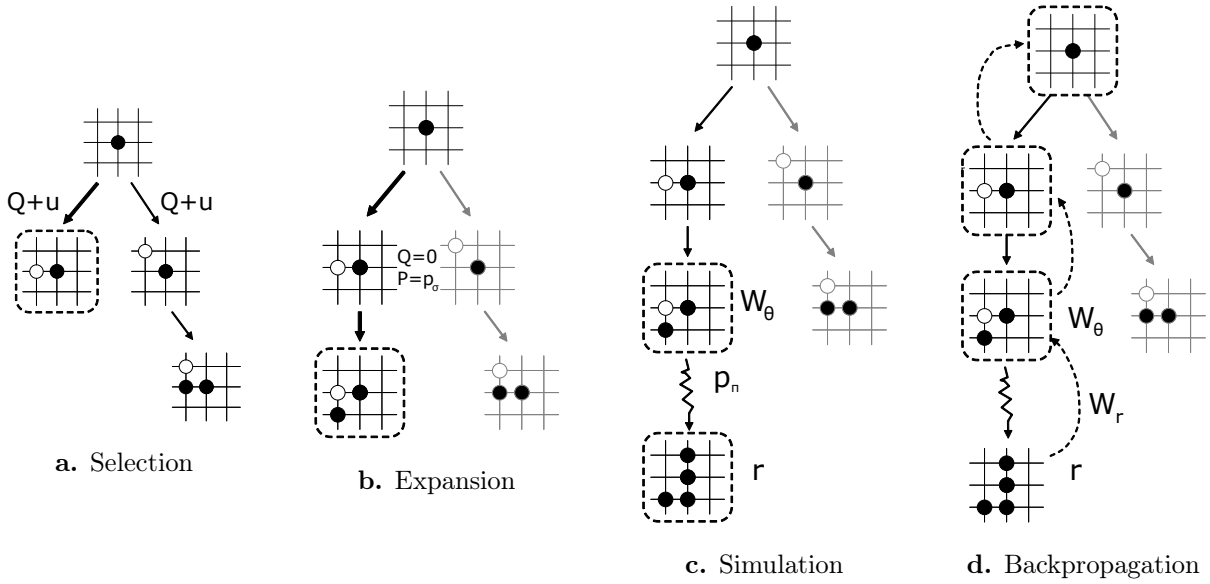

Figure 4: Monte Carlo Tree Search in AlphaGo. In the selection stage the decision is influenced by the prior probability from the SL network (a). The value of the node, Q, is calculated by both the rollout outcomes W_r and the value network estimations W_θ. The simulations are guided by the rollout classifier (c) and their result is then backpropagated (d).

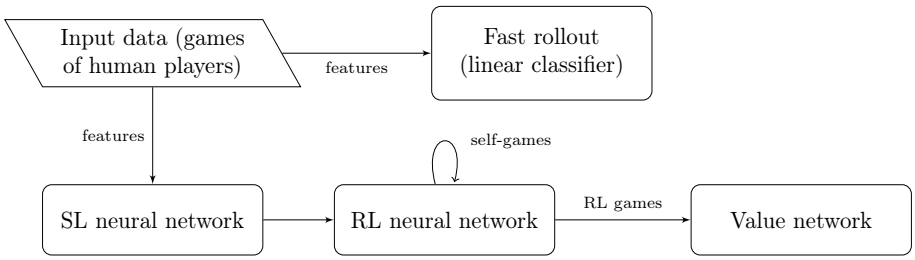

Figure 5: The learning pipeline of AlphaGo. SL denotes supervised learning, RL reinforcement learning

5 The Story of AlphaGo's Success

AlphaGo was launched as a research project by the Google team DeepMind in 2014. In October 2015 it became the first computer program to beat a human player in the full-sized game (on 19×19 board). That opponent was Fan Hui, a European Go champion, who possessed a rank of 2 *dan* out of 9 possible [2].

After that AlphaGo went on to play against one of the best players of Go, Lee Sedol, in March 2016. AlphaGo was able to defeat him winning four out of five games. That was a great milestone in the development of the AI.

In December 2016 Go community was baffled by an unknown player "Master" on the Go online server, who by the end of month managed to win over 30 top-ranked professionals without losing. In January 2017 DeepMind admitted that it was a new version of AlphaGo.

Some people say that because all the best players were defeated, it can be considered the total victory of AlphaGo. Others, however, claim that AlphaGo still can lose in an official full-length game. This argument will probably dissipate in April 2017 when AlphaGo will take an official match against Ke Jie, who is considered the last person capable of defeating AlphaGo.

6 Conclusion

In this article we addressed the phenomenon of AlphaGo - the first AI to master the game of Go. Let us recap the contents. After defining the rules of Go we explained that the computers solve the game by traversing the game tree. However, as the game tree for Go is huge, one has to use a statistical approach, such as MCTS. Then we outlined several improvements to MCTS and after that we went on to see that AlphaGo strengthens MCTS with the use of convolutional neural networks.

[2]Dan is a master rank of Go players the highest being 9

So what lessons did we learn from AlphaGo? The power of AlphaGo is undoubtedly vast, it managed to make a revolution in the state of the art AI. The number of games AlphaGo played to train itself was more than all the people have ever played [WZZ+16]. However, AlphaGo is not a breakthrough technology, all the methods that it uses have been known and developed for a long while. Therefore, we can claim that AlphaGo is a consequence of the recent research in computer Go. Moreover, there are still some enhancements that could be made to AlphaGo [Rai16].

Arguably, one of the greatest advantages of AlphaGo is that it uses general-purpose algorithms, not bound to the Go domain. AlphaGo is an example which has shown that such a complicated problem as Go, could be solved with the state-of-the-art techniques. Deep learning has been successfully used in image and natural language processing, biomedicine and other fields. Perhaps the pipeline introduced by the authors of AlphaGo after some modifications can be used in one of them [WZZ+16].

References

[Bai10] Hendrik Baier. Adaptive playout policies for Monte-Carlo Go. *Mém. de mast. Institut für Kognitionswissenschaft, Universität Osnabrück*, 2010.

[BC01] Bruno Bouzy and Tristan Cazenave. Computer Go: an AI oriented survey. *Artificial Intelligence*, 132(1):39–103, 2001.

[Bou04] Bruno Bouzy. Associating shallow and selective global tree search with Monte Carlo for 9*9 Go. In *Computers and Games*, volume 3846 of *Lecture Notes in Computer Science*, pages 67–80. Springer, 2004.

[Bou06] Bruno Bouzy. Move-pruning techniques for Monte-Carlo Go. In *ACG*, volume 4250 of *Lecture Notes in Computer Science*, pages 104–119. Springer, 2006.

[BPW+12] Cameron B Browne, Edward Powley, Daniel Whitehouse, Simon M Lucas, Peter I Cowling, Philipp Rohlfshagen, Stephen Tavener, DieGo Perez, Spyridon Samothrakis, and Simon Colton. A survey of Monte Carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in games*, 4(1):1–43, 2012.

[Brü93] Bernd Brügmann. Monte Carlo Go. Technical report, Citeseer, 1993.

[CBSS08] Guillaume Chaslot, Sander Bakkes, Istvan Szita, and Pieter Spronck. Monte-Carlo tree search: A new framework for game AI. In *AIIDE*. The AAAI Press, 2008.

[Chi96] Shirish Chinchalkar. An upper bound for the number of reachable positions. *ICCA JOURNAL*, 19(3):181–183, 1996.

[CI07] Xindi Cai and Donald C. Wunsch II. Computer Go: A grand challenge to AI. In *Challenges for Computational Intelligence*, volume 63 of *Studies in Computational Intelligence*, pages 443–465. Springer, 2007.

[CM07] Pierre-Arnaud Coquelin and Rémi Munos. Bandit algorithms for tree search. pages 67–74, 2007.

[Cou06] Rémi Coulom. Efficient selectivity and backup operators in Monte-Carlo tree search. In *Computers and Games*, volume 4630 of *Lecture Notes in Computer Science*, pages 72–83. Springer, 2006.

[Cou07] Rémi Coulom. Computing "elo ratings" of move patterns in the game of Go. volume 30, pages 198–208, 2007.

[CS15] Christopher Clark and Amos J. Storkey. Training deep convolutional neural networks to play Go. 37:1766–1774, 2015.

[Dah99] Fredrik A Dahl. Honte, a Go-playing program using neural nets. *Machines that learn to play games*, pages 205–223, 1999.

[DU07] Peter Drake and Steve Uurtamo. Move ordering vs heavy playouts: Where should heuristics be applied in Monte Carlo Go,". In *Proceedings of the 3rd North American Game-On Conference*, pages 171–175, 2007.

[FB10] Hilmar Finnsson and Yngvi Björnsson. Learning simulation control in general game-playing agents. In *AAAI*. AAAI Press, 2010.

[Fis96] George S. Fishman. *Monte Carlo : concepts, algorithms, and applications*. Springer series in operations research. Springer, New York, Berlin, 1996. With 98 illustrations (p. de titre).

[Fra57] R Frank. The perceptron a perceiving and recognizing automaton. Technical report, tech. rep., Technical Report 85-460-1, Cornell Aeronautical Laboratory, 1957. 2, 1957.

[GS07] Sylvain Gelly and David Silver. Combining online and offline knowledge in UCT. In *ICML*, volume 227 of *ACM International Conference Proceeding Series*, pages 273–280. ACM, 2007.

[HM95] Jun Han and Claudio Moraga. The influence of the sigmoid function parameters on the speed of backpropagation learning. In *IWANN*, volume 930 of *Lecture Notes in Computer Science*, pages 195–201. Springer, 1995.

[HM13] Shih-Chieh Huang and Martin Müller. Investigating the limits of Monte-Carlo tree search methods in computer Go. In *Computers and Games*, volume 8427 of *Lecture Notes in Computer Science*, pages 39–48. Springer, 2013.

[Jui99] Hugues Rene Juille. Methods for statistical inference: Extending the evolutionary computation paradigm. 1999. AAI9927229.

[KM75] Donald E Knuth and Ronald W Moore. An analysis of alpha-beta pruning. *Artificial intelligence*, 6(4):293–326, 1975.

[Kor85] Richard E Korf. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial intelligence*, 27(1):97–109, 1985.

[KS06a] Levente Kocsis and Csaba Szepesvári. Bandit based Monte-Carlo planning. In *ECML*, volume 4212 of *Lecture Notes in Computer Science*, pages 282–293. Springer, 2006.

[KS06b] Levente Kocsis and Csaba Szepesvári. Bandit based Monte-Carlo planning. In *ECML*, volume 4212 of *Lecture Notes in Computer Science*, pages 282–293. Springer, 2006.

[KSW06] Levente Kocsis, Csaba Szepesvári, and Jan Willemson. Improved Monte-Carlo search. *Univ. Tartu, Estonia, Tech. Rep*, 1, 2006.

[Lit94] Michael L Littman. Markov games as a framework for multi-agent reinforcement learning. In *Proceedings of the eleventh international conference on machine learning*, volume 157, pages 157–163, 1994.

[Mec98] David A Mechner. All systems Go. *The Sciences*, 38(1):32–37, 1998.

[MHSS14] Chris J. Maddison, Aja Huang, Ilya Sutskever, and David Silver. Move evaluation in Go using deep convolutional neural networks. *CoRR*, abs/1412.6564, 2014.

[MP43] Warren S. McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5(4):115–133, 1943.

[Mül02] Martin Müller. Computer Go. *Artificial Intelligence*, 134(1-2):145–179, 2002.

[Rai16] Tapani Raiko. Towards super-human artificial intelligence in Go by further improvements of AlphaGo. 2016.

[RTL+10] Arpad Rimmel, Olivier Teytaud, Chang-Shing Lee, Shi-Jim Yen, Mei-Hui Wang, and Shang-Rong Tsai. Current frontiers in computer Go. *IEEE Transactions on Computational Intelligence and AI in Games*, 2(4):229–238, 2010.

[Sch10] W Schubert. KGS Go server, 2010.

[SHM+16] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of Go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 2016.

[SN08] Ilya Sutskever and Vinod Nair. Mimicking Go experts with convolutional neural networks. In *ICANN (2)*, volume 5164 of *Lecture Notes in Computer Science*, pages 101–110. Springer, 2008.

[ST09] David Silver and Gerald Tesauro. Monte-Carlo simulation balancing. In *ICML*, volume 382 of *ACM International Conference Proceeding Series*, pages 945–952. ACM, 2009.

[TF06] John Tromp and Gunnar Farnebäck. Combinatorics of Go. In *Computers and Games*, volume 4630 of *Lecture Notes in Computer Science*, pages 84–99. Springer, 2006.

[Upt98] Robin JG Upton. Dynamic stochastic control: A new approach to tree search & game-playing. *University of Warwick, UK*, 23, 1998.

[VDHUVR02] H Jaap Van Den Herik, Jos WHM Uiterwijk, and Jack Van Rijswijck. Games solved: Now and in the future. *Artificial Intelligence*, 134(1-2):277–311, 2002.

[WG07] Yizao Wang and Sylvain Gelly. Modifications of UCT and sequence-like simulations for Monte-Carlo Go. pages 175–182, 2007.

[WH86] DRGHR Williams and GE Hinton. Learning representations by back-propagating errors. *Nature*, 323(6088):533–538, 1986.

[Wil92] Ronald J Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8(3-4):229–256, 1992.

[WZZ+16] Fei-Yue Wang, Jun Jason Zhang, Xinhu Zheng, Xiao Wang, Yong Yuan, Xiaoxiao Dai, Jie Zhang, and Liuqing Yang. Where does AlphaGo go: from church-turing thesis to AlphaGo thesis and beyond. *IEEE/CAA Journal of Automatica Sinica*, 3(2):113–120, 2016.